University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 2: this Friday @ 11:59 pm
  - Lab 1: tomorrow @ 11:59 pm
  - Assignment 1: Friday 2/17 @ 11:59 pm
- Lecture recordings are available on Canvas under Panopto Video
- Please use the “Request Regrade” feature on GradeScope if you have any issues with your homework grades
- TAs student support hours available on the syllabus page

Previous lecture

- Binary Search Tree

This Lecture

- Binary Search Tree
  - How to search, add, and delete
- Runtime of BST operations
- Red-Black BST (Balanced BST)
  - definition and basic operations

Reflect on the steps that you followed

- How did you add 4 to the tree?
- What steps did you follow?

10, 8, 17, 7, 5, 20, 15, 16, 4

BST: How to add?

- How to add a data item *entry* into a BST rooted at *root?*
- What if root.data.compareTo(entry) == 0?
- What if root.data.compareTo(entry) < 0?
  - Move left or right?
  - What if no child?
  - What if there is a child?
- What if root.data.compareTo(entry) > 0?
  - Move left or right?
  - What if no child?
  - What if there is a child?
- What if I tell you that you have a friend who can add into a BST.
  - How can you use the help of that friend?

Let's see the code for adding into a BST

- Available online at:
  - https://cs1501-2231.github.io/slides-handouts/CodeHandouts/TreeADT/Slides
  - The slides are under the CodeHandouts/TreeADT/slides folder in the handout repository
  - https://github.com/cs1501-2231/slides-handouts

Let's build a Binary Search Tree

- Work in groups of 2-3 students
- Add the following integers to a Binary Search Tree in **the following order**:

4, 5, 7, 8, 10, 8, 15, 16, 17, 20

Reflect on the steps that you followed

- How many comparisons did you have to make to add 20?

4, 5, 7, 8, 10, 8, 15, 16, 17, 20

Run-time of BST operations

- For add, # comparisons = d, where *d* is the depth of the new node
- For search miss, # comparisons = d, where *d* is the depth of the node if it were in the tree
- For search hit, # comparisons = 1+d, where *d* is the depth of the found node
- On average, node depth in a BST is O(log n)
  - *n* is the number of data items
  - Proof in Proposition C in Section 3.2 of Sedgewick Textbook
- In the worst case, node depth in a BST is O(n)

Let's switch to delete!

- Work in groups of 2-3 students
- In the Binary Search Tree that you built out of **the following order**:

10, 8, 17, 7, 5, 20, 15, 16, 4

- How would you delete 4?
- How would you delete 5?
- How would you delete 10?

BST: How to delete?

- Three cases
  - Case 1: node to be deleted is a leaf
    - Easiest case!
    - Pass back null to the node's parent
  - Case 2: node to be deleted has one child
    - Pass back the child to the node's parent to adopt instead of the node

BST: How to delete?

- Three cases
  - Case 3: node to be deleted has two children
    - Difficult to delete the node!
      - It has two children, so parent cannot adopt both
    - Let's try to replace the data in the node
      - We still want to maintain the search tree property
      - What are our options?
    - Replace node's data by its successor
      - largest item in left subtree
    - or by its predecessor
      - smallest item in right subtree
    - Delete the node that we selected in the previous step
      - Has to have at most one child
      - Why?

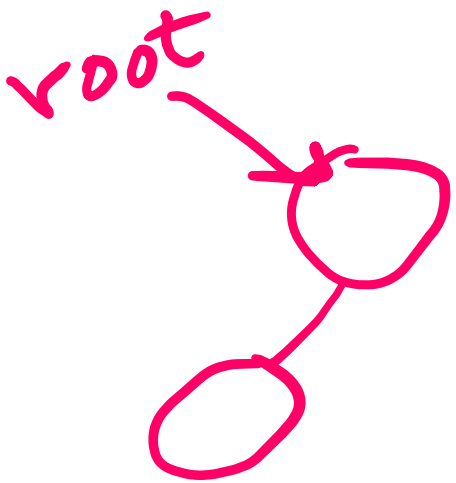
Let's see the code for deleting from a BST

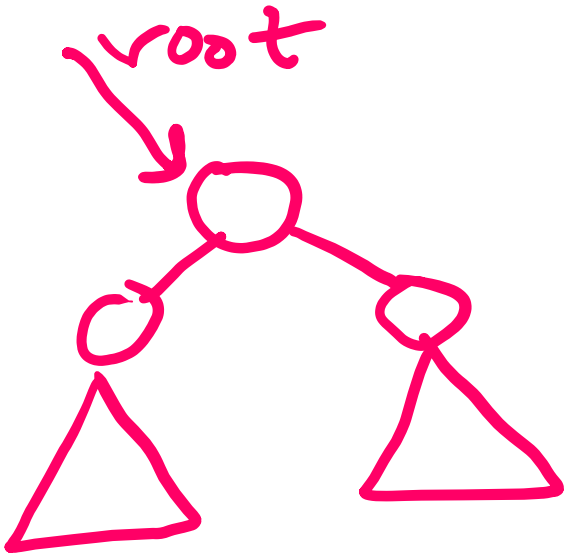
- Available online at:
  - https://cs1501-2231.github.io/slides-handouts/CodeHandouts/TreeADT/Slides
  - The slides are under the CodeHandouts/TreeADT/slides folder in the handout repository
  - https://github.com/cs1501-2231/slides-handouts

BST: delete operation

- Deleting an item requires first to find the node with that item in the tree
- Let's assume that we have already found that node
- The method below returns a reference to the root of the tree after removing its root

```
private BinaryNode<T> removeFromRoot(BinaryNode<T> root){
```





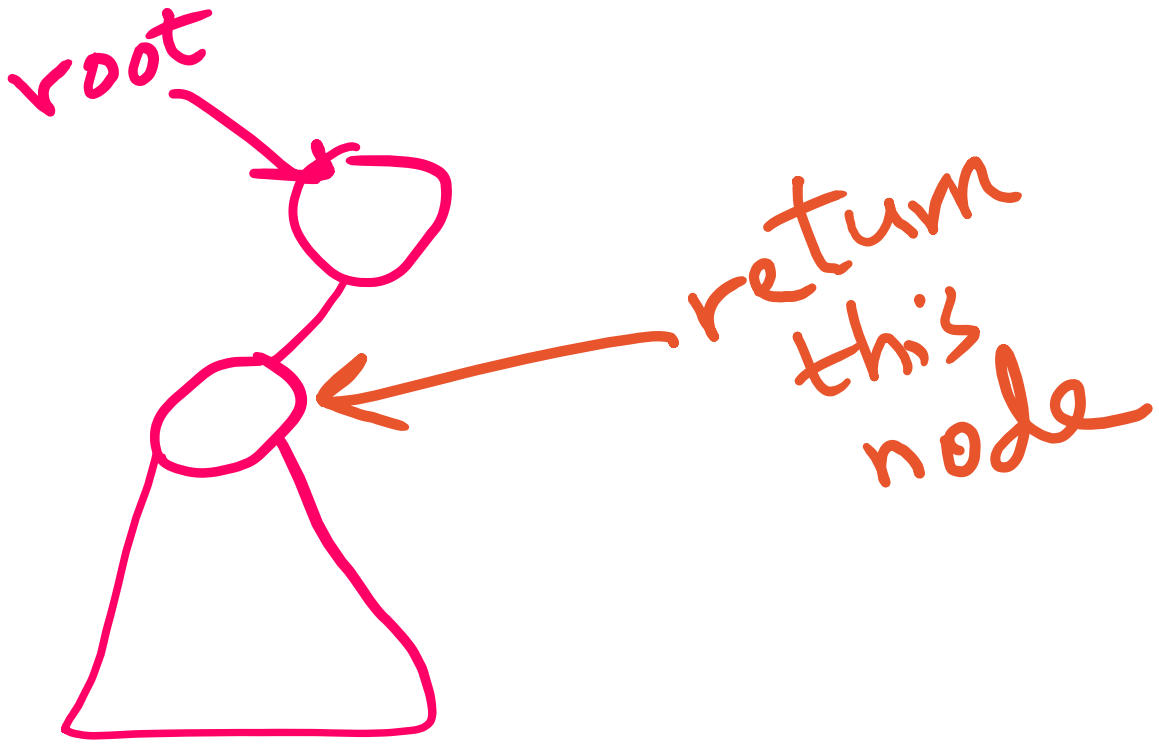
Delete Case 1: tree has only one node

```
private BinaryNode<T> removeFromRoot(BinaryNode<T> root){
```

Return null

Delete Case 1: root has one child (left or right)

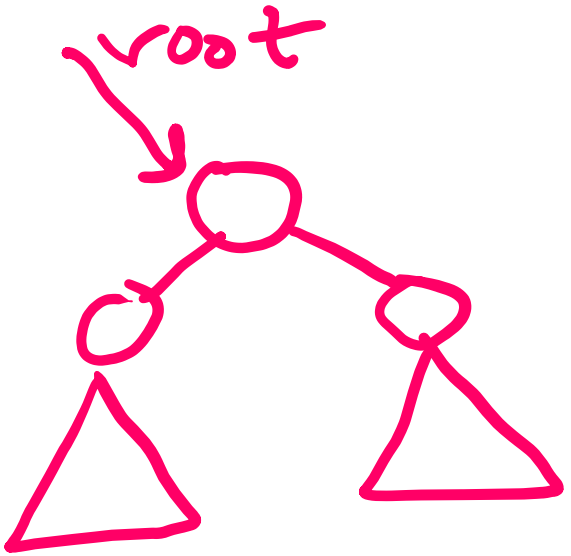
```
private BinaryNode<T> removeFromRoot(BinaryNode<T> root){
```



Return the root of the subtree rooted at the child

Delete Case 1: root has two children

```
private BinaryNode<T> removeFromRoot(BinaryNode<T> root){
```



- replace root's data by the data of the largest item of its left subtree (why?)
- remove the largest item from the left subtree
- return root

How to find largest item in a BST?

```
private BinaryNode<T> findLargest(BinaryNode<T> root){
  if(root.hasRightChild()){
    return findLargest(root.getRightChild());
  } else {
    return root;
  }
}
```

How to remove largest item in a BST?

- The method below returns the root of the tree after deleting the largest item
- If the largest item is the root of the tree, return its left child

```
private BinaryNode<T> removeLargest(BinaryNode<T> root){
  if(root.hasRightChild()){
     root.setRightChild(removeLargest(root.getRightChild()));
  } else {
    root = root.getLeftChild();
  }
  return root;
}
```

Now we need to find the node to delete

- The method below returns the root of the BST after removing the node that contains entry if found
- We also need to return the removed data item
  - How to return two things?
  - Pass a wrapper object

```
private BinaryNode<T> removeEntry(BinaryNode<T> root,
                                  T entry, ReturnObject item){
```

Wrapper Class

```
private class ReturnObject {
   T item;
   private ReturnObject(T entry){
      item = entry;
   }
   private void set(T entry){
      item = entry;
   }
   private T get(){
      return item;
   }
```

Runtime of BST operations

- Search miss, search hit, add
  - O(depth of node)
  - Worst-case: O(n)
  - Average-case: O(log n)
- Delete
  - Finding the node: O(log n) on average
  - Finding and removing largest node in subtree: O(log n) on average
  - Total is O(log n) on average
    - and O(n) in worst-case

Runtime of BST operations

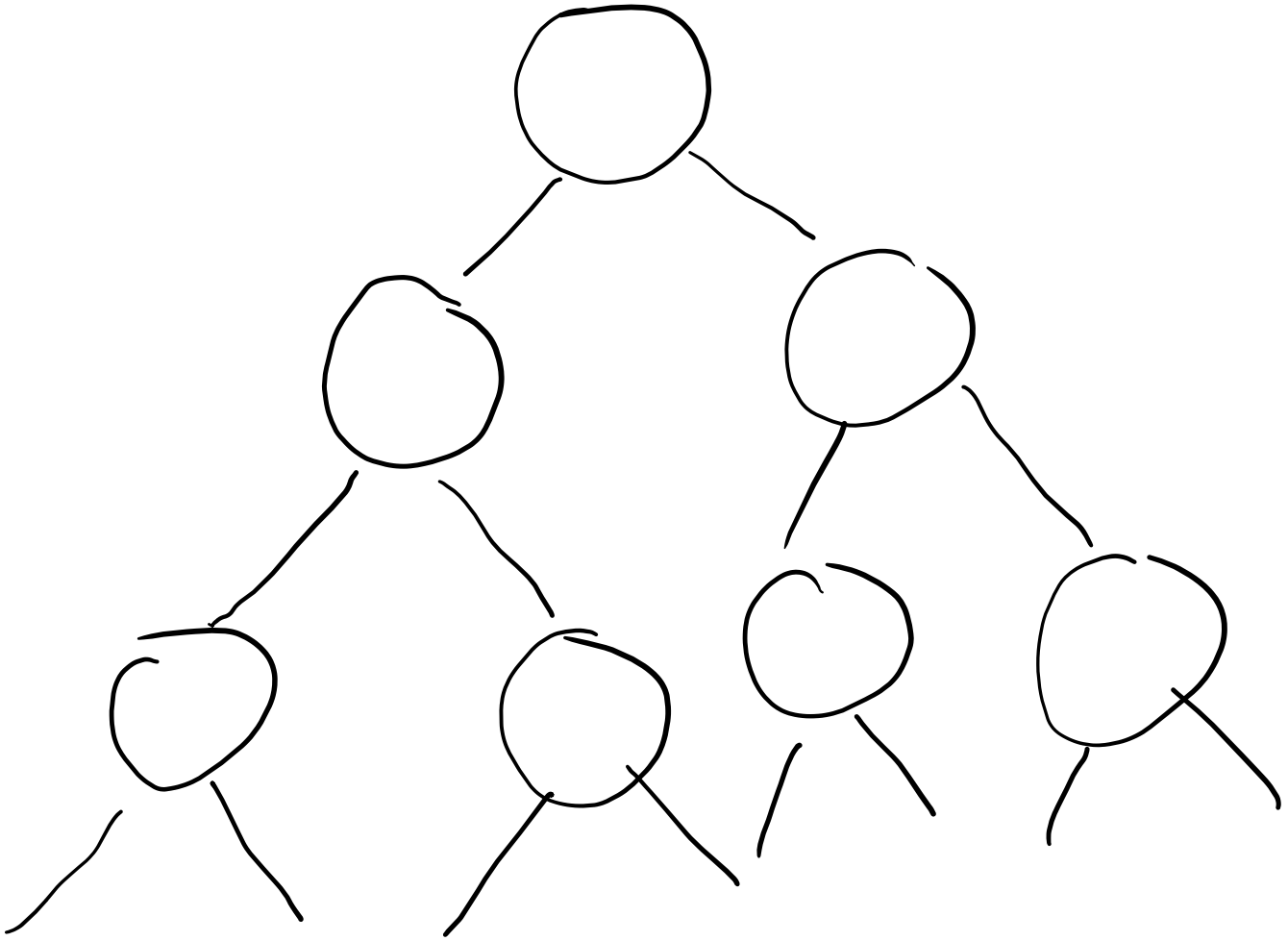
- Can we make the worst-case runtime O(log n)?
- Yes, if we keep the tree *balanced*
  - That is, the difference in height between left and right subtrees is controlled

Red-Black BST

- Definition
  - two colors for edges: red and black
  - a node takes the color of the edge to its parent
  - only left-child edges can be red
  - at most one red-edge connected to each node
  - Each leaf node has two black null-edges out of it (to the two null references)
  - all paths from root to null-edges have the same number of black edges
  - root node is black
  - ***Why?***
    - ***maximum height = 2*log n***
- Basic operations
  - rotate left
  - rotate right
  - flip color
  - ***preserve the properties of the red-black BST!***

Red-black BST example

- All black nodes → has to be a full tree
- Height = O(log(n))

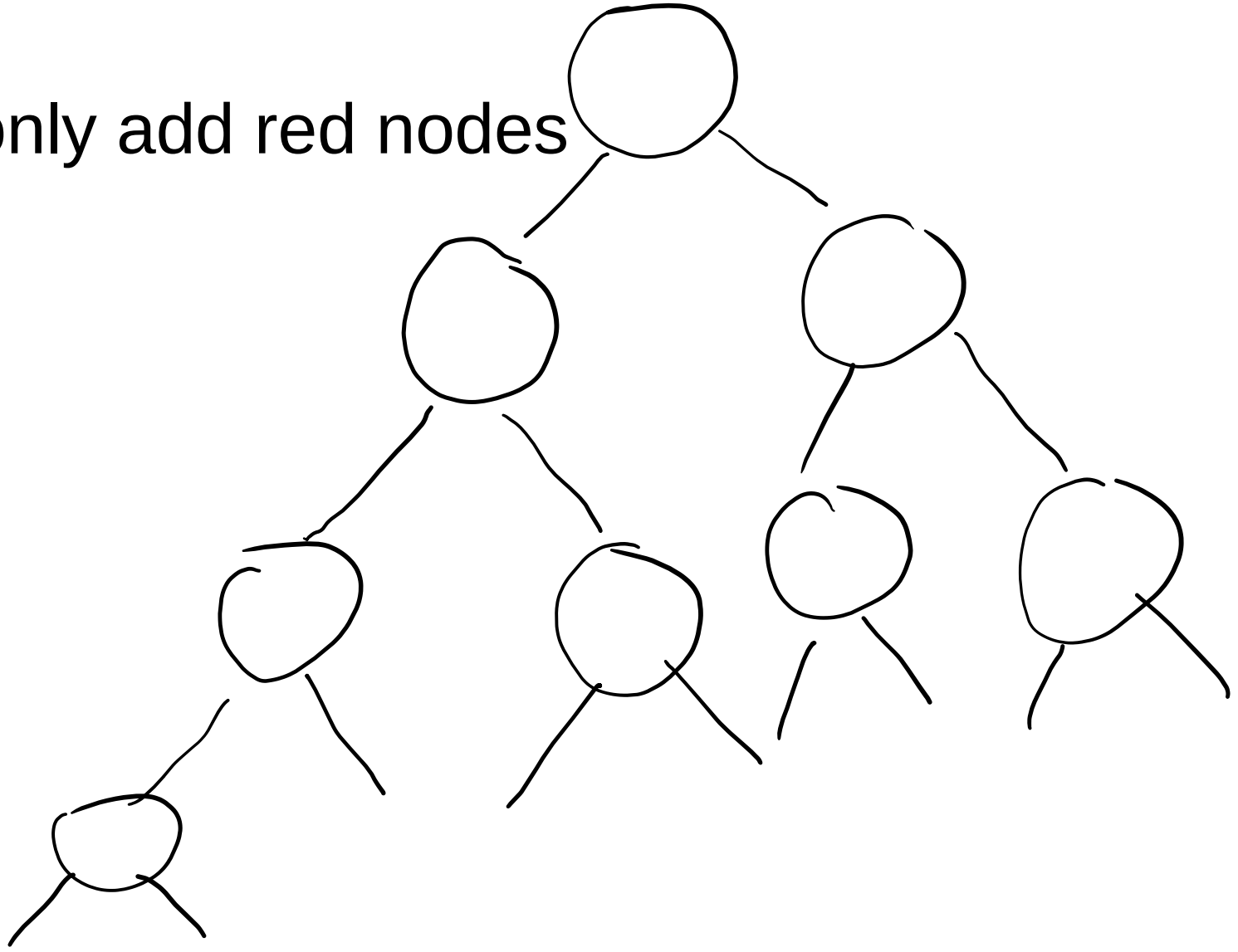


Red-Black BST example

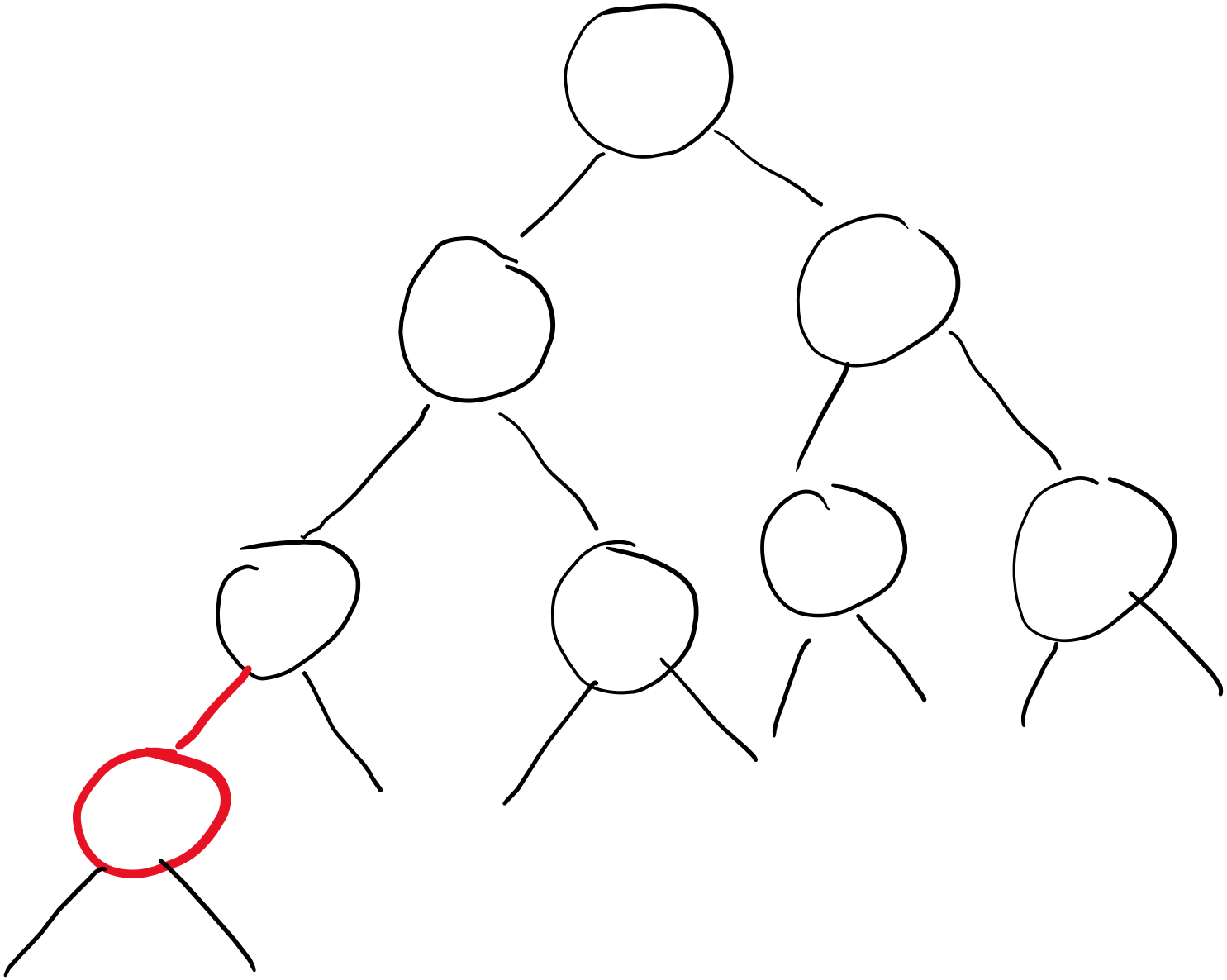
- Let’s imagine an adversary who wants to increase the height of the tree by adding the fewest number of node

Red-black BST example

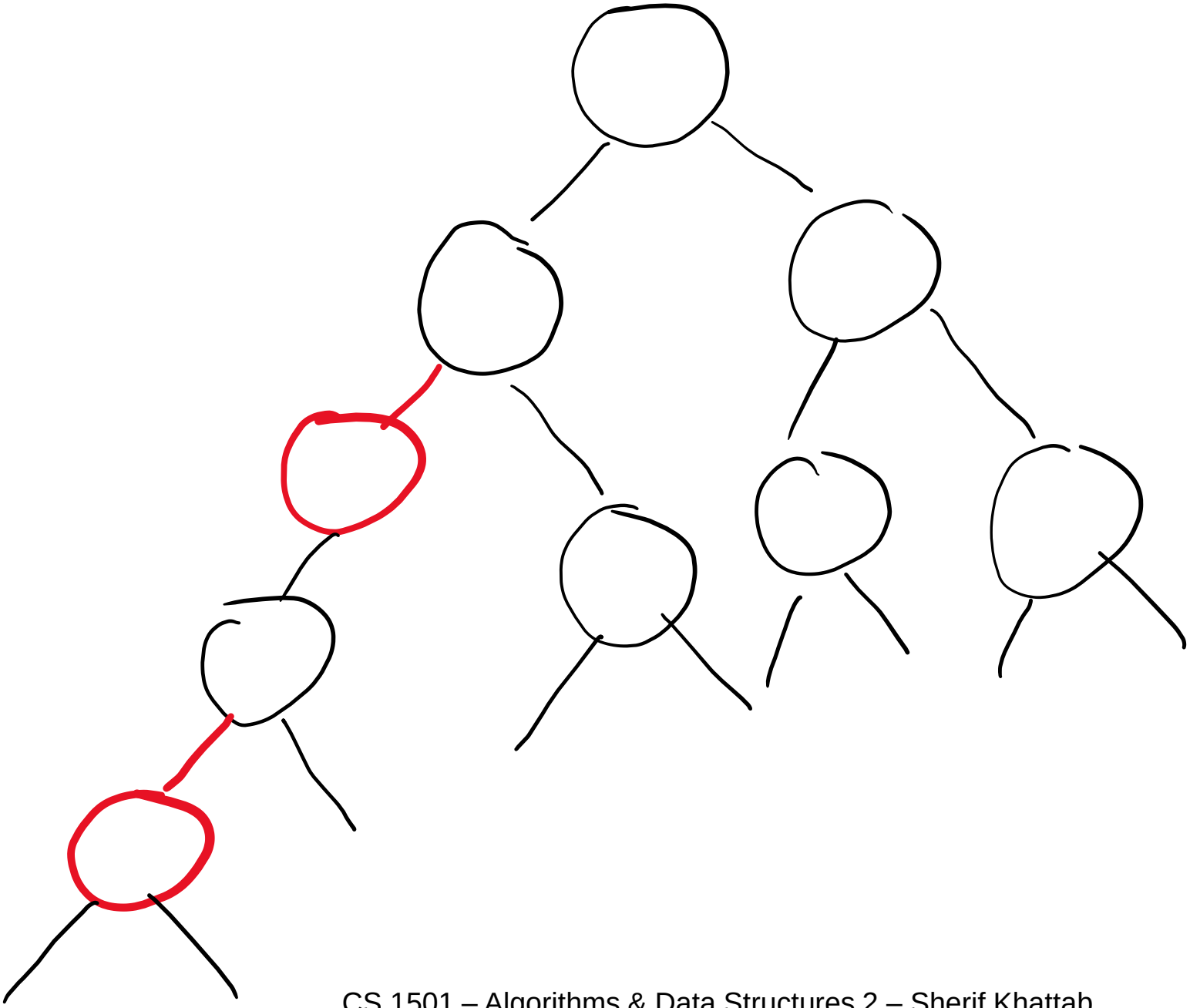
- Can the adversary add a black node?
- No! why?
- They can only add red nodes



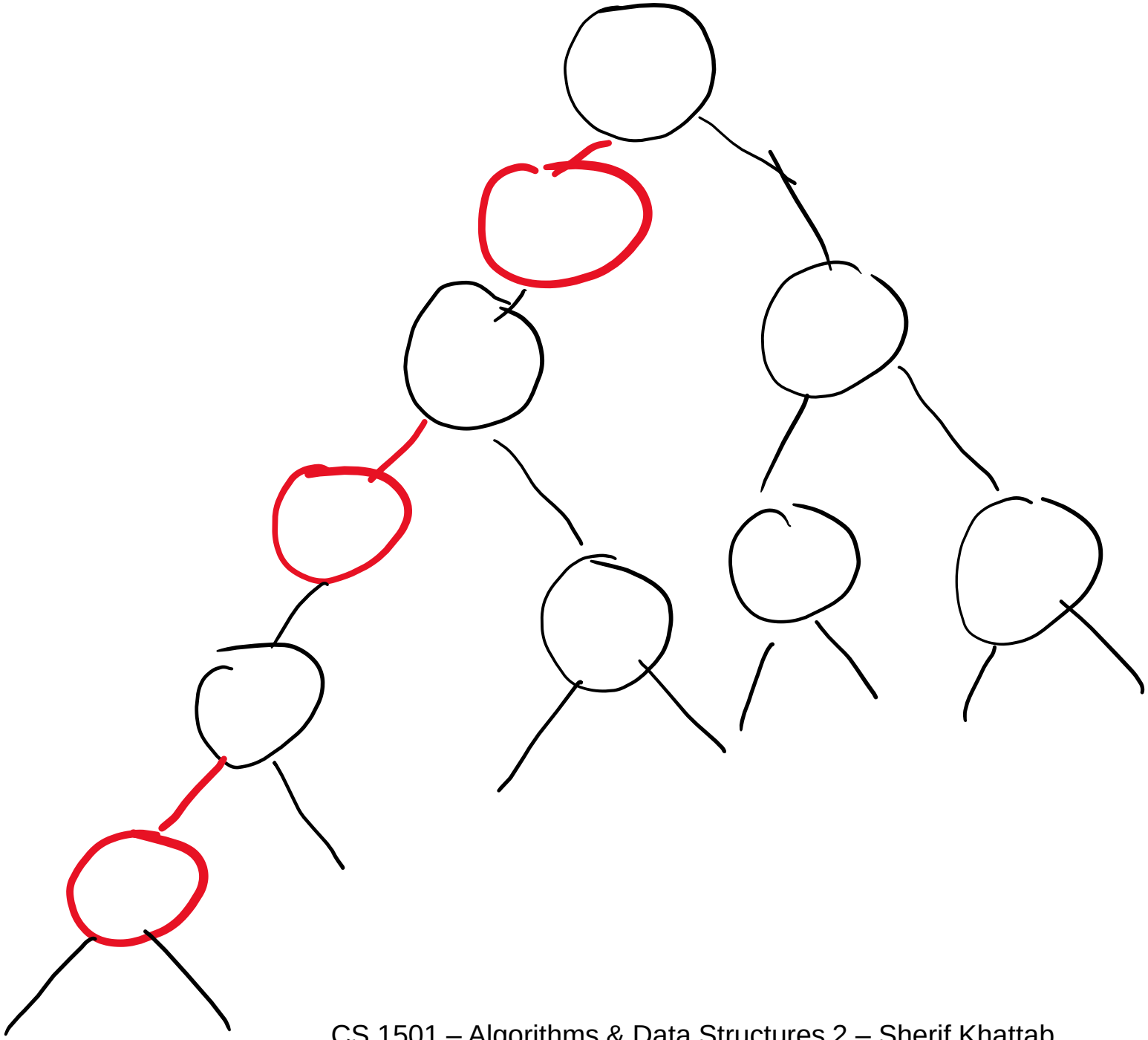
Red-black BST example



Red-black BST example

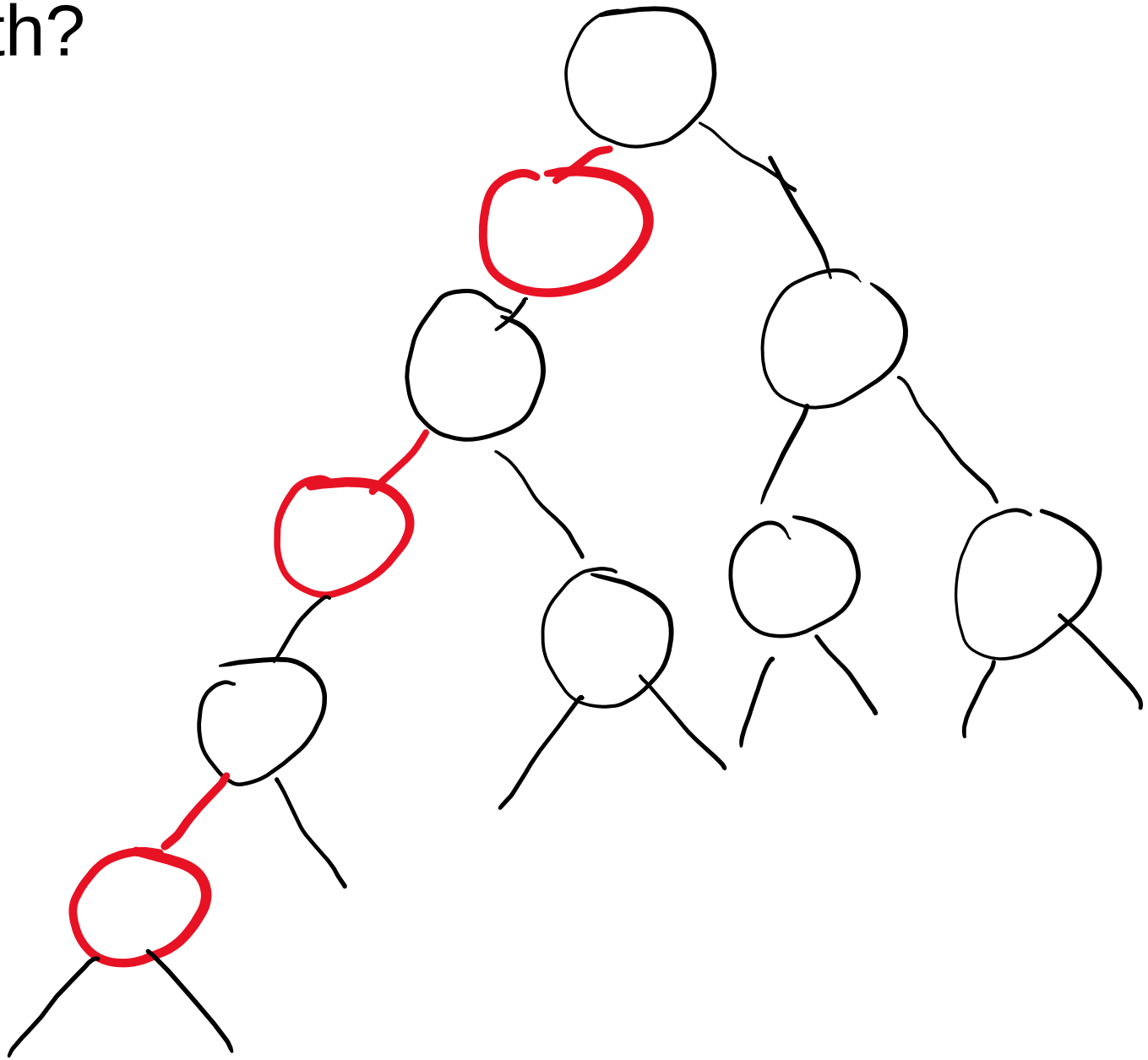


Red-black BST example



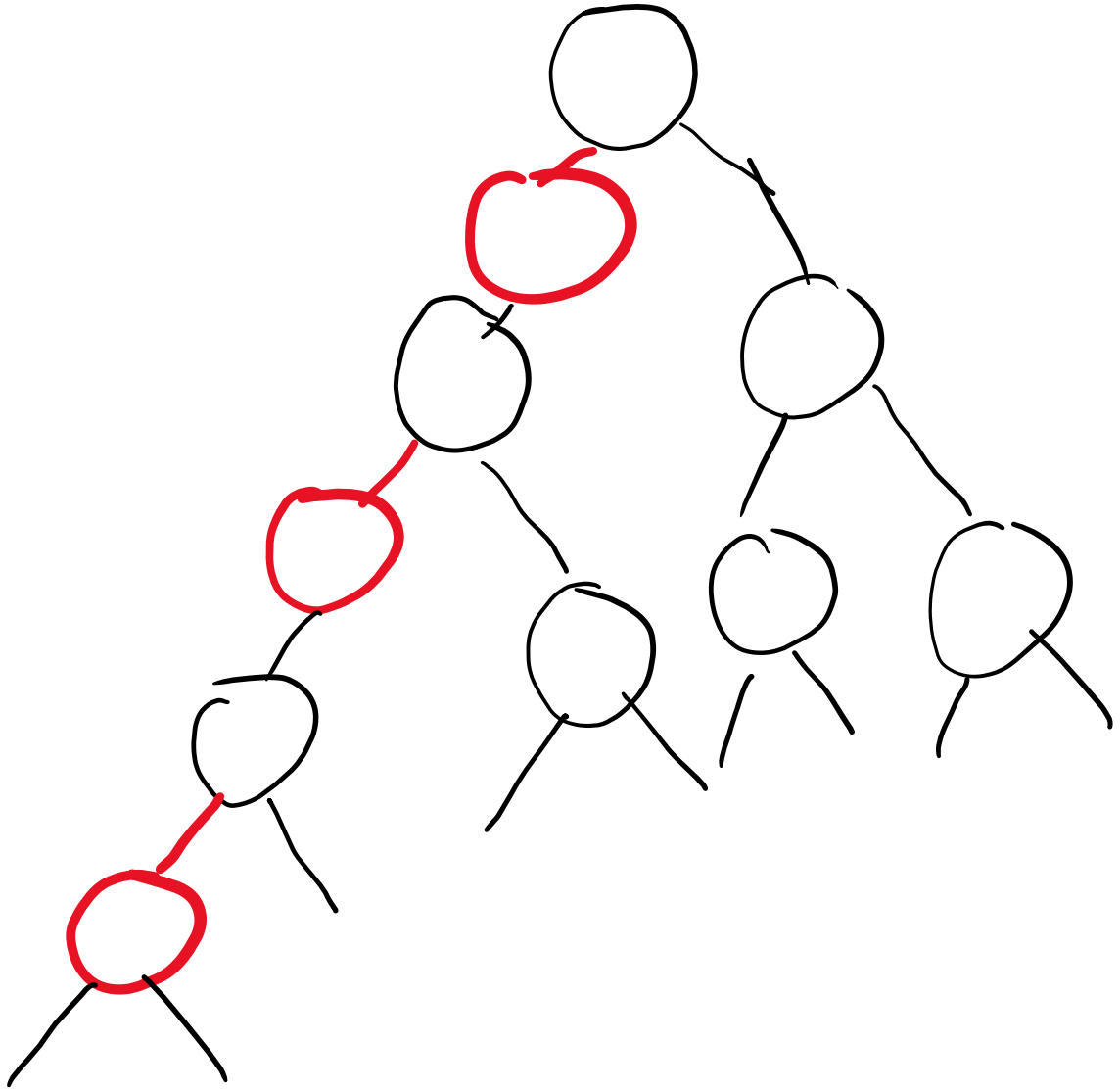
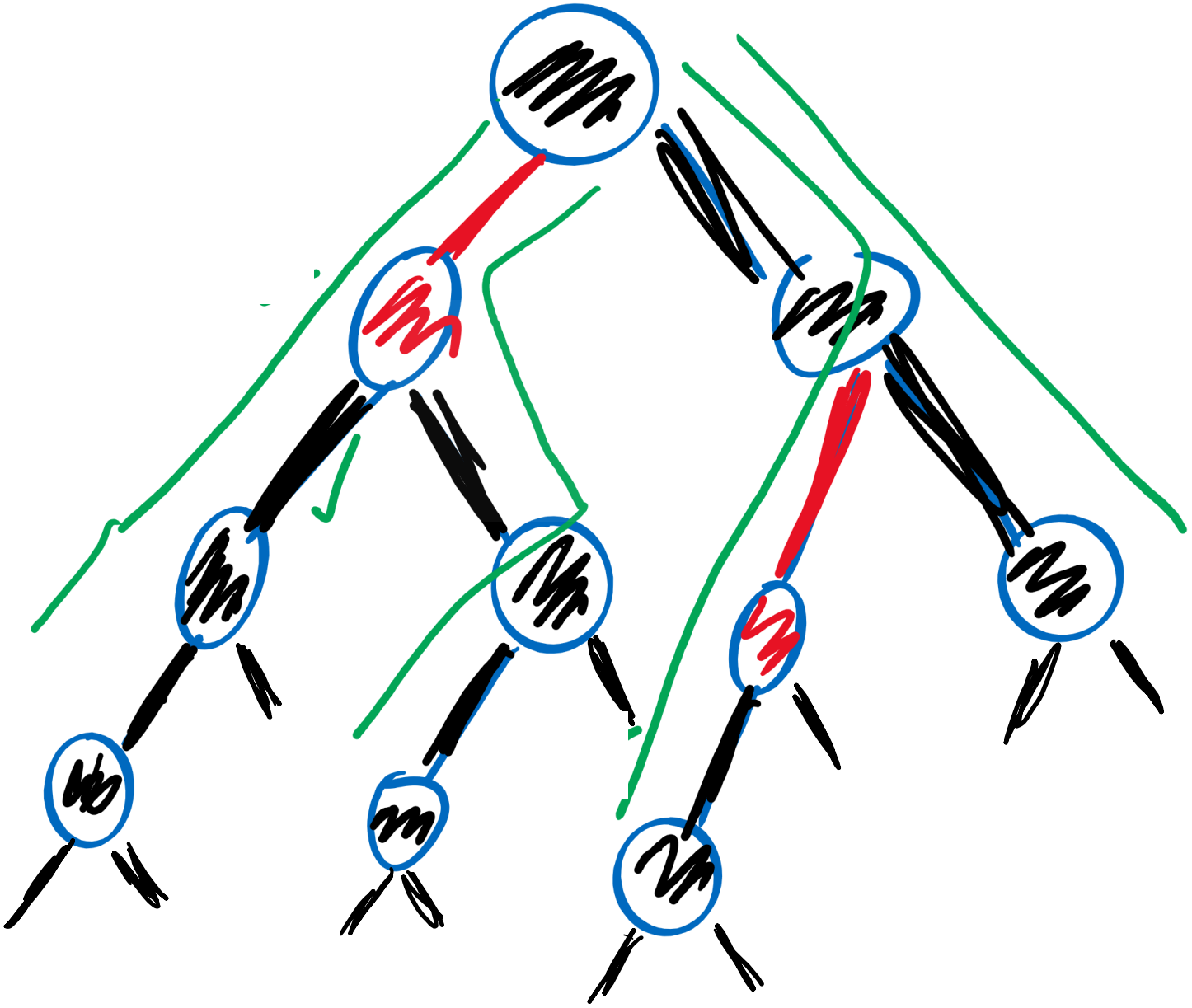
Red-black BST example

- Can the adversary add more red nodes to the left-most path?



Red-Black BST Example

- The maximum “damage” that the adversary can do is to double the height of the full tree
  - 2* log (n)
  - still O(log n)



Red-black BST non-example